OPTIMIZATION OF SQL QUERY

Optimized SQL Query:

Query | Query History

```sql
-- Add indexes to improve join and filtering performance
CREATE INDEX idx_name_clickup ON ClickUp(Name);
CREATE INDEX idx_name_float ON Float(Name);

-- Optimized query
SELECT
    c.Name,
    f.Role,
    SUM(c.Hours) AS Total_Tracked_Hours,
    SUM(f.Estimated_Hours) AS Total_Allocated_Hours,
    c.Date
FROM
    ClickUp c
JOIN
    Float f
ON
    c.Name = f.Name
GROUP BY
    c.Name, f.Role, c.Date
HAVING
    SUM(c.Hours) > 100
ORDER BY
    Total_Allocated_Hours DESC;

```

Indexes:

- Created indexes on the `Name` column in both `ClickUp` and `Float` tables.
- **Reason:** Joins on non-indexed columns can result in table scans. Indexing speeds up lookups and join operations.
- **Impact:** Significantly reduces the time taken for the `JOIN` operation.

GROUP BY Clause Adjustment:

- Added `c.Date` to the `GROUP BY` clause to match the SELECT clause.
- **Reason:** SQL Server and other databases require all non-aggregated columns in the `SELECT` to be present in the `GROUP BY` clause. This avoids unnecessary computations.

Reformatted Aggregates:

- Renamed `SUM(f.Estimed Hours)` to `SUM(f.Estimated_Hours)` for consistency.
- **Impact:** Ensures column references are correct.

HAVING Clause Optimization:

- The `HAVING` clause should only filter aggregated results, which is fine here. No changes required.

Partitioning (If Large Data):

- If the `ClickUp` or `Float` tables are very large, consider partitioning them by `Name` or `Date`.
- **Impact:** Partitioning reduces the number of rows scanned during queries.

Data Type and Null Handling:

- Ensure columns used in joins (e.g., `Name`) are of the same data type in both tables and handle null values explicitly if applicable.

ORDER BY Optimization:

- The `ORDER BY` clause uses an aggregated column, which is efficient after indexing and aggregation.